Fundamentals of Artificial Intelligence [H02C1a]

Xinhai Zou (r0727971)

Contents

1 Intro to AI

1.1 Overview

AI starts from **1955** in Bell Telephone Laboratories. What is AI today?

- Planning & Scheduling
- Declarative Programming
- Robotics
- Computer vision
- Machine learning
- language processing
- Human computer interaction
- theorem proving

Sometime, we must make a decision when faced with insufficient information. Being rational means maximizing your expected utility.

1.2 Summary

1. One summer was not enough to shape AI -> but many significant advances have been made since!
2. Movie AI, News AI and Research AI differ
3. Modern AI systems combine 'reasoning' with 'learning' -> this course: 'resoning' focus
4. Much to say about ethics and fairness of AI systems

2 Rational agent

2.1 Overview

- Rational agent
  - Agent: an entity that perceives (from env.) and acts (to env.)
  - Rational: maximizing expected utility
  - Rational agent: selects actions that maximize its (expected) utility, i.e., they do the right thing.
- Different environments
  - Accessible vs. inaccessible (fully observable vs. partially observable)
    - environment is fully observed by agent?
  - dterministic vs. stochasitic
    - is the next state of the envioronment completely determined by the current state?
  - episodic vs. sequential
    - can the quality of an action be evaluated within **an episode** (perception + action)?
  - static vs. dynamic
    - can the environment change while the agent is deliberating?
    - dynamic - not turn-based game, static - mostly turn-based game
  - discrete vs. continuous
    - is the value discrete (nominal) or continuous (numerical)
  - single vs. multi-agent
    - which entities have to regarded as agents? (competitive or cooperative scenarios?)

Example

- Reflex agents
  - choose (respond) action only based on current state
  - do not consider the future consequences of their actions
- Planning/ Goal based agent
  - choose action based on consequences of actions (future)
  - consider how the world will be

Table 1: Examples using different environments

Env.	Observable	Agents	Deterministic	Episodic	Static	Discrete
Crossword puzzle	Fully	Single	Deterministic	sequential	Static	Discrete
Chess with a clock						
Poker						
Backgammon						
Taxi driving						
Medical diagnosis						
Image analysis						
Part-picking robot						
Refinery controller						
English tutor						
Pac-Mac	Observable	Multi	Stochasitic	Sequential	Dynamic	Discrete

2.2 Summary

1. an agent is something that perceives and acts, being rational is to obtain maximal expected value. An rational agent will take the action that maximizes its expected performance given the percept sequence and its knowledge of the environment.

2. an agent program maps from a percept to an action

   (a) reflex agents respond based on current state

   (b) goal-based agents respond based on current and future state

   (c) learning agents improve their behaviour over time

3. some environments are more demanding than others

3 State space

3.1 Overview

How to represent a problem in machine? For a basic search problem, there are five ingredients.

1. States: represent the states in a clear way
2. Actions: define the actions you can execute and change the world state, normally $\delta : X \to Y$
3. Initial state: what is the initial state, the **starting** point
4. Goal formulation: define which states you want to end
5. Specification of the search cost: the execution costs

Take Pac-man - path-finding as an example, which is shown below:

- States: (x, y), where x is x-axis and y is y-axis
- Initial state: starting point (x_0, y_0)
- Actions: different actions
  - east: $f_{east} : (x, y) \to (x + 1, y)$
  - west: $f_{west} : (x, y) \to (x - 1, y)$
  - north: $f_{north} : (x, y) \to (x, y + 1)$
  - south: $f_{south} : (x, y) \to (x, y - 1)$
- Goal test: goal point/state (x^*, y^*)
- Path cost: each step cost 1 step unit, only have <specific number> state unit

Sometimes, we need a check-loop detection. We have two different structure for state-space representation: graph and search tree.

- graph
  - For graph, during searching, we often keep track of the already visited nodes to avoid loops and redundancies.
- search tree
  - nodes show states, each node in the search tree is an entire PATH in the state space graph.

Some key issues:

- definition of state space and transition model

- choose implicit graph or implicit tree
- search forward or backward
- use optimal solution or any solution
- decompose difficult problem to simpler problem (problem reduction/ decomposability)

For forward and backward search, the definition is shown as belwo. The efficiency can be improved based on different branching factor. However, backward is not always possible, e.g., playing chess.

1. Forward: start from initial state and search towards goal
2. Backward: start from the goal state and search toward initial state

3.2 Summary

1. Before an agent can start to perform on searching problem, 5 ingredients are important:
   (a) State
   (b) Initial state
   (c) Goal state
   (d) Action
   (e) Cost

4 Uniformed search

4.1 Overview

Some terminologies for searching problem in graph:

1. node expansion: generating all successor node considering the availble actions (the whole process)
2. explored nodes: nodes that have already been expanded
3. frontier/ fringe: set of all nodes available for expansion (candidate for expansion)
4. search strategy: defines which node is expanded next (the strategies)

4.2 Graph search

Graph search, frontier = <s,a>, <s,b>, its basic algorithm:
Input:
a graph,
a set of start nodes,
Boolean procedure *goal(n)* that tests if n is a goal node.
frontier := $< s >$, s is a start node
while *frontier* is not empty:
select and **remove** path $< n_0, ..., n_k >$ from *frontier* **for every** neighbor n of n_k
if *goal(n)*
return $< n_0, ..., n_k, n >$
else add $< n_0, ..., n_k, n >$ to *frontier*
end while

For loop detection, the loop-detection algorithm will be:
Input:
a graph,
a set of start nodes,
Boolean procedure *goal(n)* that tests if n is a goal node.
frontier := $< s >$, s is a start node
while *frontier* is not empty:
select and **remove** path $< n_0, ..., n_k >$ from *frontier* **for every** neighbor n of n_k
if *goal(n)*
return $< n_0, ..., n_k, n >$
else if $n \notin n_0, ..., n_k$
then add $< n_0, ..., n_k, n >$ to *frontier*
end while

Two basic searching strategies for selecting nodes to explore: depth-first search and breadth-first search.

4.3 DFS and BFS

4.3.1 Depth-First Search (DFS)

Its selecting strategy is **LIFO = "Last In First Out"**, we work with **stack**.

- Time complexity: $O(b^m)$
  - with b: branching factor, m: maximum depth
- Space complexity: $O(b \times m)$
  - with b: branching factor, m: maximum depth

The property of DFS:

- Time complexity: $O(b^m)$
- Space complexity: $O(b \times m)$
- Completeness: is complete if finite m and prevent loop
- Optimal: no guarantee, it finds the leftmost solution

4.3.2 Breadth-First Search (BFS)

Its selecting strategy is **FIFO = "First In First Out"**, we work with **queue**.

- Time complexity: $O(b^s)$
  - with b: branching factor, s: shallowest-solution depth
- Space complexity: $O(b^s)$
  - with b: branching factor, s: shallowest-solution depth
- Completeness: no matter loop or not, it is complete
- Optimal: optimal only if **cost between each node = 1**

4.3.3 Comparison of DFS & BFS

DFS uses less memory, but needs loop detection; while BFS use more memory, but does not need loop detection. More details are shown as below:

- Depth-First Search
  - Space is restricted
  - Many solutions exist, solutions are long

- no infinite paths

- Breadth-First Search
  - Space is not a problem
  - Solution containing fewest/shallowest depth

4.4 Iterative Deepening

Iterative Deepening is a combinatino of DFS and BFS, the idea is to get advantages from both DFS (space complexity) and BFS (time complexity, optimal, completeness). Run a DFS with depth limit 0/1/2/3/.... (P.S.:LIFO-stack, FIFO-queue).
Furthermore, the **outter** algorithm:

procedure iterative-deepening (
  Input:
  a graph,
  a set of start nodes,
  Boolean function *goal(n)* that tests if n is a goal mode
)
depthlimit = 1
while goal not found **do**
  call depth-limited-search(graph, starting nodes, *goal(n)*, *depthlimit*)
  depthlimit = depthlimit + 1

The **inner** algorithm:

procedure depth-limited-search(
  Input:
  a graph,
  a set of start nodes,
  Boolean procedure *goal(n)* that tests if n is a goal node,
  depthlimit: natural number
)
frontier := $< s >$: s is a start node
while *frontier* is not empty:
  select and **remove** first path $< n_0, ..., n_k >$ from *frontier*
  if $k <$ *depthlimit*
    for every neighbor n for n_k
      if *goal(n)*
        then return $n_0, ..., n_k, n$
      else add $n_0, ..., n_k, n$ to start of *frontier*
end while

The property of iterative deepening (ID) is similar to BFS, the only difference is the memory usage which is better than BFS ($O(b \times s) \geq O(b^s)$). *Time complexity almost is the same as BFS, the only difference is ID needs iterative*

reconstruction each limit of 0,1,2,3 needs to start from start node (root) and to rebuild the search tree

- Time complexity: $O(b^s)$
  - with b: branching factor, s: shallowest-solution depth
- Space complexity: $O(b \times s)$
  - with b: branching factor, s: shallowest-solution depth
- Completeness: no matter loop or not, it is complete
- Optimal: optimal only if **cost between each node = 1**

4.5 Bi-directional search

Forward and backward search together. But it is not always possible *because backward is not always known*. The algorithm is shown below:
Input:
a graph,
a set of start nodes,
Boolean procedure *goal(n)* that tests if n is a goal node.
$frontier_0 := < s >$: s is a start node
$frontier_1 := < g >$: g is a goal node
$i := 0$;
while $frontier_1$ and $frontier_0$ not empty:
$i := i + 1$ mode 2; $j := i + 1$ mod 2; (swap i and j)
select and **remove** first path $< n_0, ..., n_k >$ from $frontier_i$
for every neighbor n of n_k in direction i
if n occurs in path $< n'_0, ..., n'_l >$ of $frontier_j$ with l the length of n in $frontier_j$
return solution based on $< n'_0, ..., n'_l >$ and $< n_0, ..., n_k >$
else add $< n_0, ..., n_k, n >$ to end of $frontier_i$
end while

The property of Bi-directional search:

- Time complexity: $O(2 \times b^{\frac{d}{2}})$
  - with b: branching factor, d:maximum depth
- Space complexity: $O(2 \times b^{\frac{d}{2}})$
  - with b: branching factor, d:maximum depth
- Completeness: yes
- Optimal: yes <- **why?**

4.6 Uniform Cost Search (UCS)

BFS finds the shortest path in terms of number of actions. However, it does not find the least-cost path. We will now consider the cost between 2 nodes.
The formula: $cost(< n_0, n_1, ..., n_k >) = \sum_{i=1}^{k} cost(< n_{i-1}, n_i >)$
SELECT becomes **"priority queue, select element with lowest cost first"**
Two possibilities in graph searching: first version and last version.

1. First version: after selecting first lowest-cost node, done.
2. Last version: after checking every lowest-cost node, then done (more optimal).

Then we can have a further optimization: **Branch-and-bound principle:** when a goal is found along a path with cost C, prune all paths on the frontier with cost > C, where these cost can never be optimal since their cost has already > C.
The properties of UCS are:
The property of Bi-directional search:

- Time complexity: $O(b^{1+\frac{C^*}{\epsilon}})$
  - with b: branching factor, C^*: solution cost, arc cost (cost between 2 neighbor nodes) at least ϵ
- Space complexity: $O(b^{1+\frac{C^*}{\epsilon}})$
  - with b: branching factor, C^*: solution cost, arc cost (cost between 2 neighbor nodes) at least ϵ
- Completeness: yes (minimum arc cost is positive, $\epsilon > 0$)
- Optimal: yes

Remember: UCS explains increasing **cost contours**.

- Pros:
  - Complete and Optimal!
- Cons:
  - Explores option in every "direction", just like BFS
  - No information about goal location, only accumulation of cost (advanced algorithm should have advanced function/info providing)

4.7 Summary

1. The difference between each algorithm: **which node on the frontier will be explored next**.
2. Search algorithm is judged by its properties: **time complexity, space complexity, completeness, optimality**.

5 Informal search

a